

Liquid

智能合约编程语言新探索

微众银行区块链编著

2021年3月



VISIT...

LANZAROTE
Caliente.COM

I 目录

一、引言	01
二、智能合约编程语言概况	03
2.1 智能合约的基本概念	03
2.2 智能合约的核心价值	03
2.3 常见智能合约编程语言	04
三、智能合约编程语言面临的挑战	05
3.1 安全性急需加固	05
3.2 效率瓶颈凸显	06
3.3 开发者期望更佳体验	06
3.4 编程模式需满足多样化场景	07
四、新型智能合约语言 Liquid	08
4.1 设计思路	09
4.2 总体架构	12
4.3 Liquid 优势	14
4.3.1 安全	14
4.3.2 性能	17
4.3.3 体验	18
4.3.4 定制能力	18
五、总结与展望	21
致谢	22
参考文献	23



一、引言

2020 年 4 月 20 日，国家发展改革委首次明确新型基础设施建设（简称新基建）的范围，将区块链视为新基建的核心技术和自主创新的重要突破口。在近期发布的国家“十四五”规划纲要中，区块链也被列入数字经济重点产业，迎来发展“加速度”。具体到产业内容上，纲要明确指出推动智能合约、共识算法、加密算法、分布式系统等关键技术的创新。

作为连接区块链底层技术与现实业务的枢纽，智能合约在推动分布式应用繁荣发展的过程中扮演着重要角色。而在智能合约的开发过程中，其编程语言是开发者表达业务逻辑的抽象工具，也是构筑分布式应用的重要基石。

目前，业界已有 Solidity 语言、Move 语言及 DAML 等常见的智能合约编程语言。然而，不断多样化、复杂化的应用场景给智能合约编程语言提出全新挑战：进一步重视数据隐私，智能合约下辖的数据需能够明确权限归属；分布式、不可篡改的执行环境要求智能合约具备更强的鲁棒性；日渐扩大的服务规模要求智能合约能够更加高效运行；为了提升开发效率，开发过程需要对开发者更加友好；不断涌现的跨链等新型计算范式要求能够直接提供原生抽象。常见的智能合约编程语言在这些方面均存在不同程度的局限性，制约了分布式应用的进一步发展。

微众银行区块链着眼于现有挑战，结合在区块链底层核心技术以及分布式商业应用落地的探索实践，提出涵盖安全（**S**ecurity）、性能（**P**erformance）、体验（**E**xperience）及定制能力（**C**ustomization）四个方面的“SPEC”智能合约编程语言设计规范。

在“SPEC”设计规范的基础上，微众银行区块链推出全新的智能合约编程语言 Liquid。

在安全方面，Liquid 支持使用加密原语对数据进行隐匿，从而确保数据安全性；支持在智能合约的编译期及运行期全方位进行安全检测，从而提升逻辑鲁棒性。

在性能方面，Liquid 通过优化执行引擎并结合并行化等技术，让智能合约的执行效率取得长足进步。

在体验方面，Liquid 提供丰富的周边工具及资源，让智能合约的开发过程敏捷流畅。

在定制能力方面，通过宏扩展技术，Liquid 能够在语言层提供分布式协作、跨链协同等编程模型，帮助开发者更加便捷地实现业务需求。

本文致力于以区块链智能合约为研究对象，对智能合约及编程语言的基本概念、存在的问题以及研究现状进行系统性梳理，并介绍 Liquid 的设计理念及关键实现细节，供智能合约开发者交流探讨。微众银行区块链秉承多方参与、资源共享、友好协作和价值整合的理念，将 Liquid 项目完全向公众开源，并在 FISCO BCOS 开源社区中成立了智能合约编译技术专项兴趣小组（CTSC-SIG），欢迎广大企业及技术爱好者踊跃参与 Liquid 项目共建。



二、智能合约编程语言概况

2.1 智能合约的基本概念

智能合约的概念最早由 Nick Szabo^[1] 于 1996 年提出，所谓“合约”，是指合作的多方为达成一致所共同签订的协议，里面记录了涉及权利与义务的条款及执行条款的前置条件，其效力由第三方保证；而“智能”便意指自动化及可编程。区块链上的智能合约可以理解为在区块链上签订的协议，它以程序片段的形式存储于区块链上，根据设定的条件自动执行，无需依赖可信第三方。智能合约的参与方无需担心协议失效，因为一旦被部署至区块链上，智能合约的内容便不可被篡改。智能合约的参与方也无需担心协议不被履行，因为当达到预设条件时，区块链系统便会在分布式环境下对智能合约进行确定性执行，并根据给定输入获取对应的输出，从而保障合约执行结果有效。相比传统合约，智能合约具有如下优势^[2]：

- (1) 低风险：部署在区块链上的智能合约不能被任意篡改，且由于区块链上的交易被冗余备份在整个分布式系统中，因此这些交易都可被溯源，从而可以抑制经济诈骗等恶意行为；
- (2) 低成本：智能合约存储在区块链上，在分布式环境中自动执行，无需依赖可信第三方，因而可以降低引入第三方带来的额外成本；
- (3) 高效率：由于智能合约只要达到预设的条件就会自动执行，因此可以节省业务流程中参与方之间、参与方与第三方之间的交互时间成本。

2.2 智能合约的核心价值

数据作为重要的生产要素，是数字经济的重要驱动力，其流动性决定了数字经济发展水平。区块链技术作为数据流动的承载者，能够为社会带来深刻的生产关系变革。区块链中数据流动的具象表现便是智能合约中的代码逻辑，因而智能合约承担了协作、连接的功能，是推动“信息互联网”向“价值互联网”转型的重要推手。智能合约的出现，使得区块链的应用场景从最初的加密货币领域扩展到了更广泛的领域，如物联网应用和公共部门管理等^[3]。

以公共部门管理为例，智能合约的引入可以减少信息诈骗的风险、帮助构建透明的个人信用信息系统。例如，投票系统可以充分利用区块链技术防篡改、多中心化的技术优势，搭建司法存证仲裁链，将所有投票记录和结果的数据摘要记录在链上，在保障用户隐私的情况下，促使每笔投票结果被准确记录且无法被篡改。同时，每笔存证均有权威仲裁机构参与链上司法存证，从法律角度确保了投票结果的证据保存，最终保障了全过程投票数据的司法有效性和真实可信。

在日益全球化的今天，高效协作、多方协同式的工作已成为常态，但由于各国监管政策、贸易环境、语言等多有不同，协作效率的提高仍是各界努力的目标。基于区块链技术的智能合约应用，可以跨越不同的主体，能够在促进数据共享、优化业务流程、降低运营成本、提升协同效率等方面发挥所长，为多方协作提供高效解决方案。

2.3 常见智能合约编程语言

比特币脚本语言是智能合约编程语言的雏形，通过编写 UTXO 实现锁定脚本和解锁脚本。比特币脚本语言的设计极其简单，除了条件流程控制外，没有循环和复杂的流程控制功能。这种有限的执行环境和简单的执行逻辑有利于验证可编程货币的安全性，预防恶意攻击者利用脚本漏洞发起攻击。但是，现实中的应用场景往往十分复杂，无法用简单的比特币脚本语言来实现。

为满足复杂的应用需求，以太坊推出了静态类型智能合约编程语言 Solidity。Solidity 语言支持继承、库和复杂的用户定义类型，语法接近 JavaScript，是目前使用最广泛的智能合约语言，具有活跃的社群与丰富的周边生态。基于 Solidity 的语言基础，研究者在安全和效率方面尝试改进，提出了 Vyper 和 Fe 等智能合约语言。

Move 语言是 Facebook 发布的 Libra 中使用的智能合约编程语言。Move 语言没有过多追求应用的通用性，而是将技术路线聚焦在数字资产这一常见的应用类别，以解决实际问题、便于工程实现而展开。Move 将货币、金融衍生品等金融资产统一抽象为资源，并通过静态分析及动态检查等手段，对资源的使用方式进行严格限制，以防资源被不恰当地使用。

DAML (Digital Asset Modeling Language, 数字资产建模语言) 是专为多方金融业务协作而设计的领域特定语言。DAML 提供了面向合同的编程模型，能够方便地在代码中表达合同签署及权利授予等现实中的概念，同时还内置防御性检查机制，能够防范对合同的越权使用及恶意攻击，从而帮助开发人员可靠地实现复杂的跨企业工作流程。



三、智能合约编程语言面临的挑战

智能合约作为区块链的上层抽象，是构建新型基础设施必不可少的技术。然而，随着区块链技术与实际应用的深入结合，目前智能合约编程语言在安全性、运行效率、用户体验、编程模型等方面都暴露出一定的局限性，本节将对这些局限性进行详细阐述。

3.1 安全性急需加固

对于数据安全而言，当前区块链系统仍然面临着数据归属不明确的风险。在传统公有链系统的设计中，所有参与者都可以获得完整的交易数据并进行有效性检验，从而可以在没有法律管制的非信任网络中实现彻底的去中心化。但是，这不符合我国对数据安全监管的要求。尤其在金融、供应链等许多领域，对于保护交易隐私有着更高的要求，在使用区块链技术时，更需严格遵守相关的隐私及监管规定。

尽管目前已有一些加密手段可以加密链上数据，但是加密数据的生产、分发、使用仍然存在较高门槛，且随着产业数字化的普及，区块链必须要支持多样化的场景。为顺应这一要求，智能合约编程语言必须实现数据的所有权确定及隐私的选择性披露，对数据的产生、收集以及使用进行全流程管理，从而提升数据保护安全性，降低数据确权难度。

对于智能合约的内容安全性而言，当前的智能合约编程语言不能完全保证智能合约内容的安全性。由于不安全的智能合约会破坏区块链的稳定性并侵犯用户的资产安全，且开发者不易对已部署在区块链上的智能合约进行修复，因此确保智能合约的内容安全性至关重要。据互联网安全公司白帽汇安全研究院发布的《区块链产业安全分析报告》显示，2011 年到 2018 年 4 月，全球范围内因智能合约安全事件造成的经济损失高达 12.4 亿美元（约合人民币 80.17 亿元）。其中最著名的合约攻击事件是 The DAO 事件，其造成超过 6,000 万美元的经济损失，最后只能通过硬分叉解决。当前，开发无漏洞智能合约需要开发人员预见一切可能发生的恶意行为并设置相对应的措施，但对于大多数开发人员而言，这项要求较高。

3.2 效率瓶颈凸显

由于共识机制是区块链系统赖以稳定运行的关键基础，因此要求区块链网络中各个节点在执行智能合约时，针对相同的输入，必须产生一致的输出。若智能合约直接运行于实际硬件之上，由于硬件规格的不同，可能会导致计算结果不一致，因此区块链系统中一般采用虚拟机机制来提供一个对外独立的沙盒执行环境，并在虚拟机中完全摒除文件、网络等能够引入随机性的要素。智能合约需要先编译至中间字节码，然后由虚拟机读入并将字节码解释至本地硬件执行。同时，为确保产生一致的代码执行顺序，虚拟机环境中往往也不允许直接使用并行化等技术。

面对复杂场景时，上述种种限制使得智能合约的运行效率捉襟见肘。以使用范围最广的以太坊虚拟机 EVM 为例，其运行效率低下的原因主要来自两方面：一是由于虚拟机的中间层特性以及为防止过度消耗计算资源而采取 Gas 计费规则，其开销较高；二是因为其统一的 256 位机器码设计，导致合约代码执行效率进一步降低。

随着产业加快数字化进程，数据新基建快速发展，区块链未来需要支撑的应用场景规模也将越来越大，只有运行高效的智能合约才能应对大规模应用场景的挑战。

3.3 开发者期望更佳体验

智能合约编程语言的用户体验可以分别从狭义和广义两个角度进行理解。狭义上，用户体验取决于集成开发环境（IDE）、智能感知工具、调试器及测试工具等是否完善、好用；广义上，用户体验取决

于围绕语言构建的社区以及周边文档、课程等资源是否活跃、丰富。狭义的用户体验负责“硬”，能够帮助开发者提高开发效率、优雅地完成开发任务；广义的用户体验负责“软”，有助于降低开发者的学习门槛，快速解决使用问题和激发开发者的反馈热情。只有软硬结合，才能使得智能合约编程语言生态繁荣，不断在良性发展的道路上前进。

纵观历史，任何一门成功的编程语言都必定伴随有强大且多样的周边生态。但在当前智能合约编程语言中，以使用范围最广的智能合约语言 Solidity 为例，尽管在区块链领域 Solidity 的周边工具配套较为完善，但相对于通用编程语言而言，其生态建设仍然有很长的路要走。

3.4 编程模式需满足多样化场景

目前，金融科技、政务民生、司法存证、供应链协同、税务发票、版权保护等领域均已尝试利用区块链记录和管理数据。而且，随着数据新基建的发展，越来越多的产业将会利用区块链进行数字化升级，这也意味着，智能合约的应用日趋多样化。面对这样的发展趋势，传统的面向过程或面向对象式的智能合约编程模型显得愈发捉襟见肘，例如：

- **跨链协同**：跨链协同是指对于不同区块链，进行信息交互和数据流通，但是智能合约编程语言中鲜有对跨链机制的原生支持。受限于智能合约编程模型，当前的跨链方案需要借助繁琐的桥接合约、中间路由等机制来保障跨链交易的有效性及事务性。
- **多方协作**：能够模拟现代商业中多方协作行为的编程模型变得越发重要，但是开发适用于这类场景的智能合约仍存在较高门槛：一方面，开发者需要编写复杂的数据结构与程序逻辑，来管理多方协作过程中的隐私和权限等问题；另一方面，这样的开发模式需要很高的开发和审计成本，严重影响开发效率。



四、新型智能合约语言 Liquid

微众银行区块链针对常见智能合约编程语言，进行了深入研究，结合在区块链底层核心技术和应用落地的实践经验，总结提炼了该领域现存的一些主要挑战，并基于“SPEC”设计规范，推出了全面解决智能合约编程语言安全、性能、体验及定制能力等问题的新型语言 Liquid。根据“SPEC”设计规范，智能合约编程语言需要满足：

■ 保障数据安全与逻辑安全

数据安全方面，Liquid 能够提供简单的加密原语及语法，帮助开发者便捷地在智能合约中明确数据的归属，明确数据的原始内容只能由拥有权限的用户查阅及更新。数据隐匿的实现将完全由 Liquid 负责，开发者无需了解密码学算法细节。

逻辑安全方面，Liquid 内部集成有形式化验证工具，支持开发者在合约编写运行时，进行验证规范，并在编译期通过形式验证领域的自动定理证明求解器，验证程序是否符合规范。此外，Liquid 还原生支持安全资产模型，确保链上数字资产不会被恶意使用及攻击，进一步保证资产类应用的安全性。

■ 提高智能合约自身的执行效率

配合 LLVM 优化器，Liquid 支持将智能合约代码编译为可移植、体积小、加载快的 Wasm 格式字节码，并结合 Tree-Shaking 技术及 wasm-opt 等工具，进一步优化智能合约体积。此外，Liquid 使用数据流分析技术，能够自动对合约代码进行数据依赖分析，并在合约执行时根据数据依赖关系自动启用并行执行，从而释放性能潜力。

■ 让开发者拥有友好的开发体验

Liquid 配套有丰富的开发工具，包括编辑器智能提示插件、单元测试框架、模糊测试工具、依赖包管理等，开发者只需专注于合约逻辑的实现即可，从而进一步提升智能合约开发效率。

■ 根据需求方便地定制语法及实现

Liquid 在设计上安全可控，可实现对智能合约编译过程的充分定制，因此能够根据需求快速推出定制化的编程模型，例如能够模拟商业合同的可编程分布式协作（Programmable Distributed Collaboration, PDC）模型，或用于跨链协同和外部合约调用的异步编程模型。Liquid 的定制能力特性有助于区块链底层平台的基础功能在开发过程中“开箱即用”，最大化区块链的价值。

接下来的篇幅将围绕“SPEC”设计规范，具体介绍 Liquid 的设计思路 and 关键实现细节。

4.1 设计思路

为了降低开发者的学习门槛，同时融合通用编程语言拥有庞大用户基数和成熟配套工具的优势，目前智能合约的实现无需再从零开始设计语言文法及编译器，Liquid 的工程化版本选择从 Rust 语言开始，根据 SPEC 规范，以及我们对智能合约技术的理解和把握，进行融合创新，构建出面向区块链智能合约领域的特定语言。参考该设计，后续可以发展出多语言、多平台的实现。

Rust 语言在设计上跟 C++ 类似，极其强调零开销抽象和 RAII（资源获取即初始化，Resource Acquisition Is Initialization），拥有极小的运行时，其运行效率与 C/C++ 处于同一级别，非常适合对性能要求较高的系统编程领域。Rust 语言利用强大的类型系统和独特的生命周期管理实现了编译期内存管理，在保证内存安全和线程安全的同时，使编译后的程序运行速度极快。Rust 还提供函数式编程语言的模式匹配和类型推导，让程序写起来更简洁优雅。此外，宏和基于 trait 的泛型机制让 Rust 语言拥有非常强大的抽象能力，在实际工程中，尤其是在库的编写过程中可以减少很多重复代码。

Liquid 专为区块链应用设计，能够为上层智能合约开发者提供区块链的底层功能的抽象；通过 Rust 语言中宏系统提供的元编程能力，Liquid 将自身“嵌入”在 Rust 语言中，即 Rust 语言是 Liquid 的宿主语言。元编程是一种计算机程序将代码看成数据的能力，这类程序能够像普通程序在运行时更新、替换变量一样，实现在编译期更新、替换代码，即“修改代码的代码”。通过元编程技术，Liquid 对 Rust 语言的部分语法进行了重新诠释，从而能够借助 Rust 语言来方便地表达智能合约中特有的语义。使用 Liquid 编写的智能合约，本身也是合法的 Rust 程序，因此能够使用现有 Rust 标准工具链进行开发。下列代码展示了一份简单的 Liquid 智能合约代码样例：


```
#[liquid::contract]
mod hello_world {
    use liquid::storage;

    #[liquid(storage)]
    struct HelloWorld {
        name: storage::Value<String>,
    }

    #[liquid(methods)]
    impl HelloWorld {
        pub fn new(&mut self) {
            self.name.initialize(String::from("Alice"));
        }

        pub fn get(&self) -> String {
            self.name.clone()
        }

        pub fn set(&mut self, name: String) {
            self.name.set(name)
        }
    }
}
```

在上述代码中，以“#[liquid(contract)]”属性标记的 mod 代码块封装了合约状态及合约方法的定义。在“#[liquid(storage)]”属性标记的 struct 代码块中，定义了上述合约中有一个名为“name”、类型为字符串的合约状态变量。在“#[liquid(methods)]”属性标记的 impl 代码块中，则定义了上述合约中的外部方法，其中“new”、“get”及“set”方法分别用于初始化、读取及设置“name”状态变量。

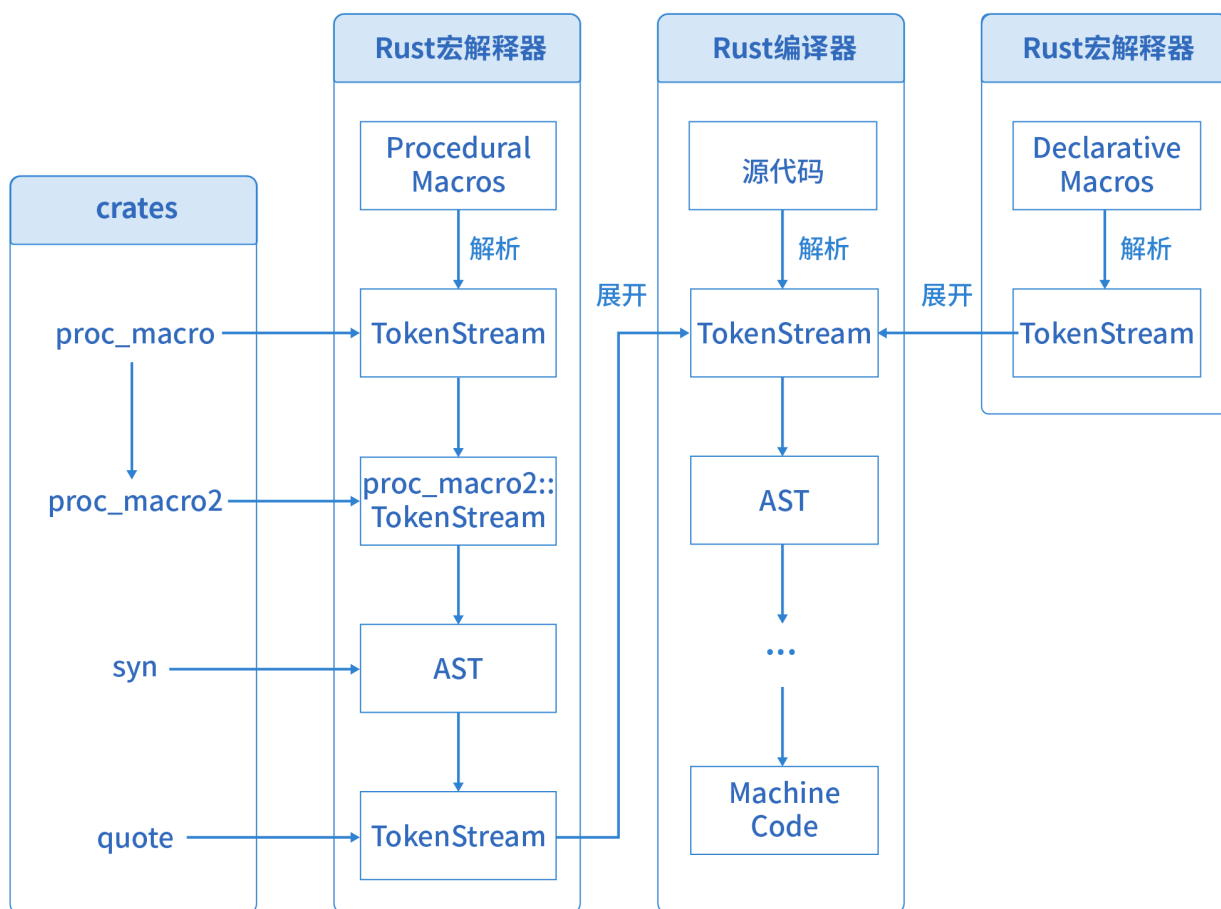


图 1 Rust 语言编译流程

上图（图 1）展示了 Rust 语言编译的整体流程。在编译阶段，Rust 编译器首先对文本格式的源代码进行词法分析，将其切割为编译器可识别的标记符（如标识符、运算符等）流。在词法分析阶段，若遇到宏代码，则 Rust 编译器会调用专门的宏解释器以解析宏代码，将宏代码展开为标记符流。Rust 语言中的宏分为三种，声明宏（Declarative Macros）、过程宏（Procedural Macros）以及派生宏（Derive Macros）。

Liquid 主要以过程宏和派生宏的形式展开工作。Liquid 中的过程宏及派生宏均以标记符流为输入，并在内部以自定义的规则对源代码进行替换。例如，在对合约中的状态变量定义时，Liquid 会分别为定义中每个状态变量生成读写区块链状态的代码，从而允许开发者以操纵成员变量的形式操纵合约状态变量，并将修饰后的代码转换成标记符流输出。最后，Rust 编译器将经过宏展开后的标记符流送入语法分析器，构建出能够表征程序结构的抽象语法树（Abstract Syntax Tree, AST），并最终生成可执行的二进制代码。

4.2 总体架构

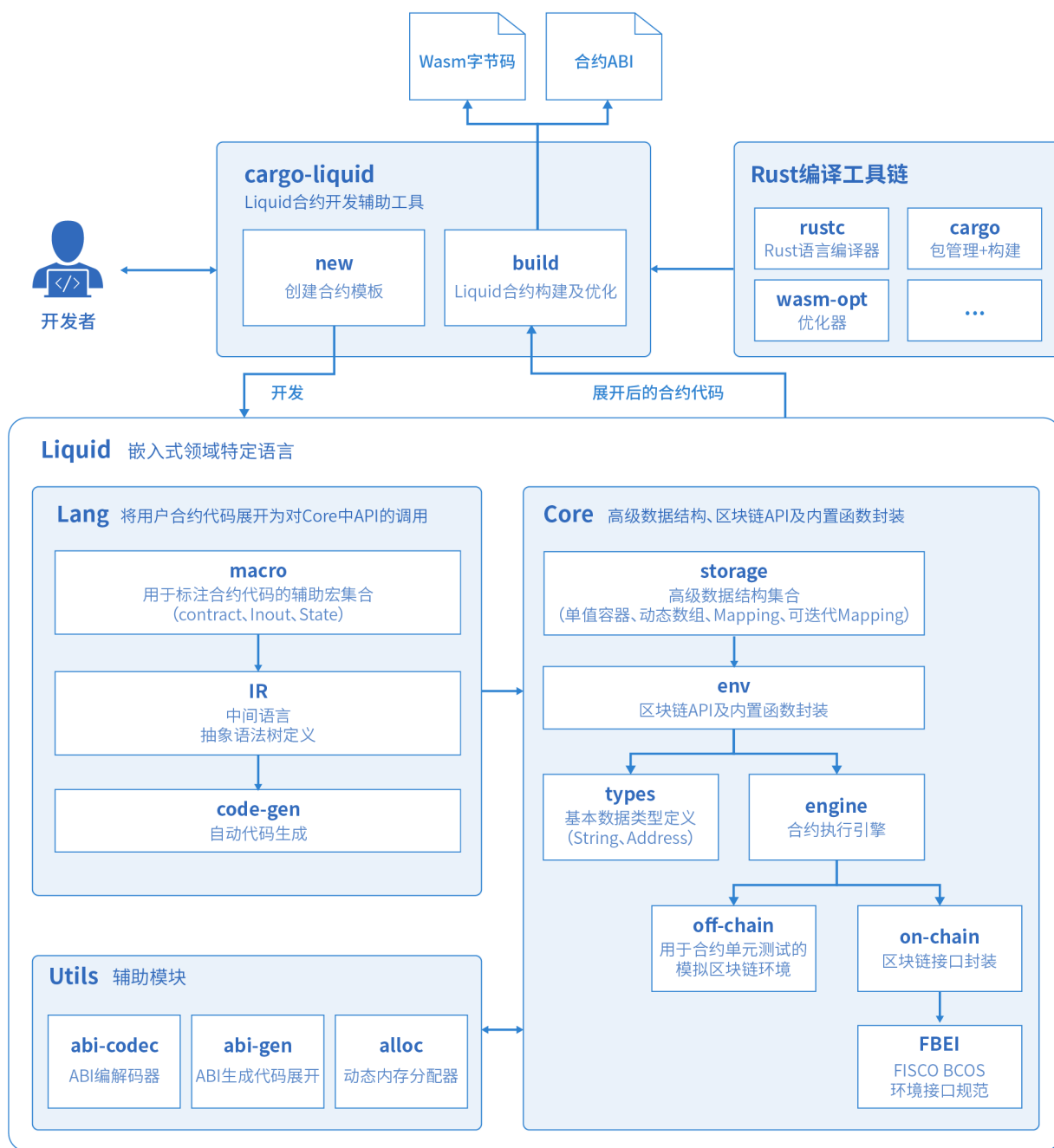


图 2

上图（图 2）展示了 Liquid 的整体架构，其中 cargo-liquid 是面向开发者的命令行辅助工具，帮助开发者创建及构建 Liquid 项目。在项目创建阶段，cargo-liquid 能够根据用户选定的项目类型以及模板，自动配置编译选项及外部依赖，并生成 ABI 生成器等辅助代码。在项目构建阶段，cargo-liquid 负责收集编译元信息并进行跨平台构建，将 Liquid 项目编译为 Wasm 格式字节码。基本构建完成后，cargo-liquid 还会使用 Tree-Shaking 算法及 wasm-opt 等工具对生成的字节码进行效率和体积上的进一步优化。

Liquid 中的 Lang 组件主要包括开发者在合约开发过程中所使用到的 contract 过程宏（用于以 mod 语法声明的智能合约）、InOut 派生宏（用于以 struct 语法定义的结构体参数类型）等，这些宏均由 macro 模块定义并导出。当构建 Liquid 项目时，Rust 语言编译器会对这些宏进行模式匹配并展开。在宏的展开过程中，IR 模块会解析开发者的代码并重新生成 AST，以对部分 Rust 语法进行重新诠释。随后，code-gen 模块会依据 IR 模块生成的 AST，调用 Core 模块中的区块链底层接口封装，展开后的代码对开发者完全透明。

Liquid 中的 Core 组件包含了开发者能够使用的区块链底层功能的实现。从底部往上的视角来看，engine 模块是 Liquid 智能合约的执行引擎，为合约运行提供最为坚实的基础。对于上层，engine 模块提供了一系列基础 API，包括用于读取链上存储的 get_storage 接口、用于写入链上存储的 set_storage 接口、用于获取当前区块时间戳的 now 接口等。对于这些接口，engine 有两种版本的实现：off-chain 版本用于在本机执行智能合约的单元测试时使用，其内部模拟了区块链特性（键值对存储、事件记录器等）并提供了测试专用的接口，帮助开发者在正式部署合约前测试合约逻辑是否正确；on-chain 版本用于智能合约在真正的区块链环境中执行时使用，由于其具体实现是由区块链底层平台完成，on-chain 中只负责对这些接口进行声明并适配即可，实现较为简单。

区块链底层接口的规范（名称、参数类型、返回值类型等）由区块链底层平台给出。对于 FISCO BCOS, 这个规范称为 FISCO BCOS 环境接口规范 (FISCO BCOS Environment Interface, FBEI)。理论上，只要接口规范、确定且底层能够提供对应的支持，便能够对接其他区块链平台，Liquid 亦如此，从而能够做到“一处编译，处处运行”。

Core 组件中的 types 模块提供了智能合约中基本数据类型的定义，如地址 (Address)、字符串 (String) 等。types 模块与 engine 模块一同构成了智能合约的执行环境，即 env 模块。storage 模块基于 env 模块提供接口，对链上状态的访问方式进行了进一步抽象。智能合约需要通过 storage 模块提供的容器类型读写链上状态。若要访问简单合约状态，则可以使用单值容器 Value；若要以下标的形式、序列式地访问合约状态，则可以使用向量容器 Vec；若要以键值对的形式访问合约状态，则可以使用映射容器 Mapping；若需要在 Mapping 的基础上根据键对合约状态进行迭代访问，则可以使用可迭代映射容器 IterableMapping。

Liquid 中的 Utils 组件涵盖了许多基础功能，主要包括用于实现合约方法参数及返回值编解码的 abi-codec 模块（此模块是 Liquid 与 Solidity 合约进行通信的关键）、用于生成 ABI 的 abi-gen 模块以及用于内存分配的 alloc 模块。其中，alloc 模块用于注册全局内存分配器，合约内所有的内存分配操作（动态数组、字符串等）都会通过 alloc 模块进行。

4.3 Liquid 优势

4.3.1 安全

在隐私保护方面，Liquid 能够更好地保护特殊数据，并控制隐私数据的访问权限。Liquid 首先使用加密原语对特殊数据进行隐匿，当用户更新数据时，使用非交互式零知识（Non-Interactive Zero-Knowledge, NIZK）证明强制证明状态更新的正确性。具体地，如果 Liquid 开发者想确保合约中的某些变量的值不会被披露在区块链上，只需要在相关的状态变量后面，用 Liquid 提供的加密原语标注能够读写该状态变量的账户地址即可，即明确数据的归属权。例如：

```
#[liquid(storage)]
struct HelloWorld {
    alice: storage::Value<address>,
    count @ alice: storage::value<u32>
}

#[liquid(methods)]
impl HelloWorld {
    pub update(&self, delta @ alice: u32) {
        count += delta;
    }
}
```

上述代码中，“alice”是一个账户地址，“count @ alice”则表示“count”状态变量仅能由“alice”所代表的账户地址访问。在编译的过程中，Liquid 会自动在合约中添加所有权校验及同态计算的代码，确保合约运行过程中数据不泄露。因此，Liquid 开发者无需亲自在合约中实现零知识证明算法，也可以保护合约的隐私。这对不具有隐私保护算法背景知识的合约开发者极其友好。

通过对现有的公有链智能合约安全漏洞的分析，可以发现 96% 的安全漏洞类别与资产转移相关^[4]，资产类合约的安全形势极为严峻。大部分公有链智能合约在资产管理方面存在的主要问题是公有链对

用户自定义资产进行了差别对待。在设计上, Liquid 重点关注资产安全, 并支持对资产运作的合规监管、隐私保护。在 Liquid 中, 资产来自实体世界、数字化领域的输入, 体现为权益、数据实体、合同承诺等多种形态。但凡被视为“资产”, 它们在链上的地位平等, 均属于线性资产模型 (Linear Asset Model, LAM) 类的合约应用, 具备同等的安全性和可控性, 体现为基于 LAM 实现的资产不允许凭空产生和销毁, 也不允许复制, 只能在严谨的业务规则、严格的权限控制、严密的安全检查、严肃的合规审计条件下, 在不同的实体间转移, 从而避免资产滥发、丢失及数量不守恒等问题。在 Liquid 中, 开发者能够根据业务需求, 面向多样化的资产, 制定灵活的管理和交易规则。如下列代码示例所示, 开发者可通过属性语法指定资产的名称、发行方、发行总量等属性:

```
#[liquid::asset(  
    issuer = "0x83309d045a19c44dc3722d15a6abd472f95866ac",  
    fungible = true,  
    total = 10000000000,  
    description = ""  
)]
```

经过广泛调研我们认为, 尽管市场中存在的资产形态与定义各异, 但万变不离其宗, 作为价值流动的载体, 资产之间往往存在着共通的特性。因而, 在接口设计上, Liquid 对资产接口的类别及数量进行了抽象及提炼, 最终将资产接口总结归纳为了三类, 通过这三类接口, 开发者能够方便、安全地实现资产类应用:

- 环境交互类: 这类接口主要用于实现资产注册、事件触发等功能。其中资产注册用于通知区块链底层, 为资产分配必要的存储与计算资源, 事件触发则是用于实现资产逻辑与业务层的触发式联动, 使得在发生资产转移等关键事件时让外界能够感知;
- 价值转移类: 这类接口主要用于转移链上资产, 可以实现将资产移交至指定账户的功能。这类接口由区块链底层设计者精心实现, 能够确保交易结果的正确性及事务性, 即交易过程中不会存在逻辑漏洞, 若交易失败也能够保证状态能够及时回滚, 不会引发资产损失;
- 状态查询类: 主要用于查询资产数量、资产发行方等元信息, 帮助用户充分掌握资产状态。

除 LAM 外，Liquid 还内置多种智能合约安全检测方法，包括单元测试、形式化验证等。

工程实践中，一般使用测试驱动开发（Test Driven Development，TDD）来保证开发过程的敏捷、高效，即在开发功能代码之前，先行编写针对函数、模块等单元的测试用例。然而在传统智能合约的开发过程中，测试链路往往较为冗长：开发者需要自行搭建模拟测试网络，再基于模拟网络进行合约部署，随后通过调用合约改变区块链状态并进行观察，以评估合约方法的逻辑正确性。这种测试方式往往无法获得快速反馈，一定程度上导致了目前智能合约中的漏洞不易被发现^[5]。为了降低智能合约单元测试的复杂性，提高合约代码的测试覆盖率，Liquid 支持在合约中编写单元测试用例，并内置完整的区块链环境模拟，使得开发者可以直接在本地方便地执行单元测试。此外，Liquid 还提供有测试专用 API，使得在运行单元测试时，能够基于这些 API 获取或改变模拟区块链环境中的状态，方便开发者对智能合约进行更深入的测试，从而进一步降低合约代码出错的几率。以前述 Hello World 合约为例，开发者能够以如下方式在合约中编写针对合约方法的单元测试用例：

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn get_works() {
        let contract = HelloWorld::new();
        assert_eq!(contract.get(), "Alice");
    }

    #[test]
    fn set_works() {
        let mut contract = HelloWorld::new();
        let new_name = String::from("Bob");
        contract.set(new_name.clone());
        assert_eq!(contract.get(), "Bob");
    }
}
```

通过对现有智能合约漏洞检测方法的深入调研，我们发现许多基于形式化验证的智能合约漏洞检测方法都依赖交互式定理证明器，例如 Isabelle/HOL 和 Coq。这些方法需要大量的人力投入，并要求相关人员具备极强的专业知识背景，因此很难被智能合约开发者广泛使用^[6]。

为了解决这个问题，让一般的智能合约开发者也可以使用形式化方法验证智能合约的正确性，Liquid 内置一种能够自动证明自定义函数属性的验证方法。该方法基于抽象解释和符号执行。用户只需要使用 Liquid 提供的属性文法为函数语句编写前置规范，例如执行某语句前要求某个变量的值必须处于指定范围类，在合约构建过程中的验证阶段，Liquid 便会自动对这些规范进行静态验证，以判断当前的代码实现是否能够满足规范中的要求，若发现可能导致规范不满足的反例，则提示开发者进行确认及修复。

4.3.2 性能

在合约运行效率方面，Liquid 使用 Wasm 引擎来负责合约的执行。Wasm 是一种可移植、体积小、加载快的字节码格式，Liquid 支持将合约代码编译为精简的 Wasm 格式字节码，从而使得智能合约能够以更高的效率在区块链系统中运行。Wasm 目前有多种高性能的虚拟机实现，理想状况下虚拟机性能可接近本地二进制代码的执行效率。另外，Wasm 最初是为浏览器的运行环境而设计，天然具备沙盒属性，方便与外界隔离，契合区块链的技术需求。再者，Wasm 设计简洁、安全，能够将可能的攻击面最小化，这一点同样符合区块链对于安全性的需求。据了解，Wasm 标准仍会不断进步，在未来还可能引入并行、多返回值、动态链接等高级特性，选用 Wasm 可持续从其中受益。

此外，Liquid 还能够支持数据流分析技术对合约中的代码进行数据依赖分析，并从中挖掘并行执行的可能性，例如下列简单的转账逻辑的实现：

```
pub fn transfer(&mut self, from: address, to: address, value: u8) {  
    self.balances[&from] -= value;  
    self.balances[&to] += value;  
}
```


Liquid 能够以 “from” 及 “to” 参数中的内容为索引，自动检测代码逻辑，修改底层存储系统中的数据状态。合约部署时，Liquid 会将其数据依赖特征发送至区块链底层平台，随后在合约执行时，执行环境能够根据交易中的数据自动调度与当前交易不冲突的交易并行执行，即并行执行交易的同时确保不会有多笔交易对相同的存储状态进行读写，进而安全地提升合约的执行性能，开发者无需顾虑并行执行过程中的同步与竞态条件等问题。

4.3.3 体验

基于 Rust 语言繁荣的发展与生态，Liquid 在体验方面，为智能合约开发带来事半功倍的效果。

首先，Liquid 拥有完善的开发库，这也得益于 Rust 语言中第三方库的发展非常迅速。通过包管理工具 Cargo，只要第三方包能够支持 “no_std” 特性，就能够在 Liquid 项目中使用。开发者仅需要在项目配置文件中声明所依赖的第三方库名、版本及特性即可在项目中使用。而且，目前在 Rust 语言生态中，越来越多的第三方库开始支持 “no_std” 特性，包括可用于 JSON 编解码的 `serde`、管理全局静态变量的 `lazy_static` 等。同时，Cargo 也为项目的打包、发布提供了一键式管理，大大减轻了开发人员的负担。

再则，Liquid 能够和现有开发工具有机结合。在开发 Liquid 项目时，开发者能够使用 GDB、LLDB 等工具对合约进行快速调试，也能够使用 `cargo-fuzz` 等模糊测试工具对合约进行深入测试，此外，还可以通过 `wasm-opt`、`wasm-strip` 等工具进一步优化字节码的体积与效率。在众工具加持下，开发者能够以更好的体验开发出更高效、健壮的 Liquid 项目。

4.3.4 定制能力

在实现机制上，Liquid 广泛地使用了 Rust 语言中基于宏的元编程技术，能够在编译阶段对由源代码生成的 AST 进行任意修饰，因而能够赋予既有语法更多的语义内涵。借助这一特性，Liquid 除了可以提供传统的智能合约编程模型外，还能够在区块链底层平台提供的接口基础上，结合现实中的业务需求定制更为简洁、强大的编程模型。例如 PDC 及跨链协同。

PDC 能够帮助开发者轻易地模拟现实商业合作中多方缔结合同的行为。在这类多方协作场景中，多涉及参与方、合同签署、权利授予等概念，而这些概念的实现往往琐碎且复杂，开发者难以驾驭。而且，这类场景中往往又对编码的正确性十分敏感，因为多个组织通过相同的智能合约进行合作，任何错误

都可能会通过链式反应逐级放大，从而造成巨大的商业损失。针对这一痛点，PDC 提供了专门针对商业合同的高级抽象，通过特定语法帮助开发者直接以自然语言的思维编写合同，例如对于简单的借据合同，在 Liquid 中可能如下方式实现：

```
#[liquid(contract)]
pub struct Debt {
    #[liquid(signers)]
    issuer: address,
    #[liquid(signers)]
    owner: address,
    amount: u64,
}

#[liquid(rights)]
impl Debt {
    #[liquid(belongs_to = "owner")]
    pub fn transfer(self, new_owner: address) -> ContractId<Debt> {
        sign! { Debt =>
            owner: new_owner,
            ..self
        }
    }
}
```

在上述代码中，由“#[liquid(contract)]”属性标记的 struct 代码块中包含了借据合同中的内容，即借据的债务人（issuer）、债权人（owner）及借款数量（amount）。代码中，债务人及债权人字段被标记为“#[liquid(signers)]”，即意味着借据合同需要债务人及债权人共同协商合同内容后方能签署，任何单边的签署都将被视为无效行为，不会在区块链中留下有效凭证。由“#[liquid(rights)]”属性的标记的 impl 代码块中则是包含了合同中权利的规定。在合同中，有一项名为“transfer”的权利，用

于转移债权，此项权利通过“#[liquid(belongs_to = “owner”)]”子句规定了只能由借据合同中的债权方行使该项权利，即其他任何参与方都将无法行使该项权利。在编译 PDC 合约时，会自动生成相应的权限检查、实例化等样板代码，且保证实现的正确性，从而帮助开发者远离繁琐的细节，因而带来开发效率上的提升。

在跨链协同方面，随着区块链适用场景日益增多，链间互操作问题亟需解决。不仅如此，因为不同链的账户状态由不同链维护，互相不可见，它们的更新也被不同链所驱动，要实现具备原子性的链间互操作面临巨大挑战。为了解决这个问题，Liquid 使用内部交易封装跨链操作，类似在同一条链上利用内部交易进行跨合约调用操作，不同的是该内部交易会带有发起跨链操作合约的签名，接着，由当前链的节点广播进行广播。同时，其它链的节点会监听跨链交易，如果识别到该交易的目的地为自己，就会像普通交易那样将交易打包上链。待交易执行完成后，所返回的数据将会以回调的形式通知原合约，以继续执行后续逻辑。开发者无需处理跨链合约调用过程中的验证和异步等，仅需以常见的 async/await 异步编程方式编写合约代码即可。



五、总结与展望

作为一种前瞻性技术，区块链一直备受各界各领域关注。随着区块链技术纳入新基建范畴，区块链将逐渐从金融行业向更大范围的行业渗透，与 5G、物联网、智能制造等共同组成新时代的技术浪潮。未来，区块链应用势必走向多元化、复杂化、协作化的发展道路。一门能够充分发挥区块链生产力、提升开发者创造性的智能合约编程语言具有重要的战略价值。

本文对智能合约及编程语言的基本概念、存在的问题以及研究现状进行了系统性梳理，提出智能合约编程语言设计的四个关键点（SPEC），即安全（Security）、性能（Performance）、体验（Experience）和定制能力（Customization）。Liquid 能够给开发者提供方便、快捷的开发体验；同时，Liquid 针对现有智能合约编程语言在安全、效率、隐私保护和跨链协同等方面的不足，针对性地提出了思考与解决方案，包括 LAM、可编程分布式协作、形式化验证、基于 NIZK 的数据确权等。

除已有的攻关与思考，作为安全可控的智能合约编程语言，Liquid 还能进一步结合技术趋势与现实需求，为智能合约创造出更多可能性。展望未来，微众银行区块链将一如既往、不忘初心，重点关注智能合约编程语言技术的发展动向，为智能合约的开发者们提供更加强大的开发平台，并通过智能合约编译技术专项兴趣小组（CTSC-SIG）及技术群等渠道为开发者打造更加繁荣友好的交流学习平台。诚邀各界伙伴协力共建，共同为加速产业的数字化升级，推动数据新基建的快速发展做出更大贡献。

2021 年技术路线

Q1 - 基础版本：完成 Liquid 基本合约编程模型、LAM、PDC 以及 cargo-liquid 的开发，并完成 Node.js SDK 的适配工作，发布 Liquid 社区体验版；

Q2 - 并行化与隐私保护解决方案：基于 Liquid 社区体验版本，完成合约自动并行化方案及基于 NIZK 的数据确权方案的语言设计及开发；

Q3 - 跨链协同解决方案：在语言层面完成与 WeCross 跨链技术的整合，研究跨系统合约调用、资产转移的异步化方案，实现跨链流程的轻量化、便捷化；

Q4 - 形式化验证方案：基于前置属性规范、符号执行、数据流分析、代码验证中间语言等技术，在 Liquid 中对合约形式化验证提供支持，帮助开发者编写更加安全、健壮的智能合约。

致谢

本白皮书的成稿，得到清华大学软件学院软件系统安全保障小组、中山大学深圳研究院区块链与智能金融研究中心相关老师和同学的大力帮助，微众银行区块链在此致以诚挚的感谢！

■ 清华大学软件学院软件系统安全保障小组（WingTecher Lab）

姜 宇 清华大学副教授、博士生导师

马福辰 博士研究生

任 萌 硕士研究生

■ 中山大学深圳研究院区块链与智能金融研究中心 (InPlusLab)

郑子彬 中山大学软件工程学院副院长、教授、博士生导师

孔雀屏 博士研究生

钟志杰 硕士研究生

叶铭熙 本科生

苏健钟 本科生

参考文献

- [1] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets", 1996, [online] Available: <http://www.fon.hum.uva.nl>.
- [2] Zheng, Zibin, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. "An overview on smart contracts: Challenges, advances and platforms." *Future Generation Computer Systems* 105 (2020):
- [3] Zheng, Zibin, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. "Blockchain challenges and opportunities: A survey." *International Journal of Web and Grid Services* 14, no. 4 (2018): 352-375.
- [4] Overview · Smart Contract Weakness Classification and Test Cases (swcregistry.io)
- [5] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, Xiang Shi. "EVMFuzzer: detect EVM vulnerabilities via fuzz testing." the 2019 27th ACM Joint Meeting ACM, 2019.
- [6] Fuchen Ma, Meng Ren, Ying Fu, Mingzhe Wang, Huizhong Li, Houbing Song, Yu Jiang. "Security reinforcement for Ethereum virtual machine." *Information Processing & Management* 58.4 (2021): 102565.

微众银行高度重视新兴技术的研究和探索，自 2015 年开展联盟链核心技术攻关和应用实践以来，已研发一整套含括底层技术、中间件、分布式数字身份、数据隐私保护、跨链、消息协作、数据治理等在内的技术方案支撑产业应用，实现全方位国产化，并持续为数字经济时代提供数据的安全存储、可信传输、协同生产等一系列数字化基础设施，释放数据生产力。

在实现区块链关键核心技术自主研发的同时，微众银行自 2017 年起主动面向全球开源，至今已开源十大区块链技术方案，牵头多方共建起最大最活跃的国产开源联盟链生态圈。生态圈内汇集 4 万余名社区用户、2000 多家企业及机构共建区块链产业生态，数百应用项目基于 FISCO BCOS 研发，其中超 120 个应用已在生产环境中稳定运行。

免责声明

在任何情况下，本白皮书中的信息或所表述的意见并不构成对任何人的投资建议，本白皮书所载的资料、工具、意见及推测只作参考之用，并非作为或被视为出售或购买证券或其他投资标的邀请或向人作出邀请。在任何情况下，白皮书的编著机构不对任何人因使用本白皮书中的任何内容所引致的任何损失负任何责任。

本白皮书主要以电子版形式分发，间或也会辅以印刷品形式分发，所有白皮书版权均归编著机构所有。未经编著机构事先书面授权，任何机构或个人不得以任何形式复制、转发或公开传播本白皮书的全部或部分内容，不得将白皮书内容作为诉讼、仲裁、传媒所引用之证明或依据，不得用于营利或用于未经允许的其它用途。如需引用、刊发或转载本白皮书，需注明出处，且不得对本白皮书进行任何有悖原意的引用、删节和修改。

所载资料仅供一般参考用，并非针对任何个人或团体的个别情况而提供。虽然我们已致力提供准确和及时的资料，但我们不能保证这些资料在阁下收取时或日后仍然准确。任何人士不应在没有详细考虑相关的情况及获取适当的专业意见下依据所载资料行事。



微众银行区块链